Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

(12)

E

INGS

1a. REPORT SECURITY CLASSIFICATION
Unclassified

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
Technical Report

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Cornell University

6b. OFFICE SYMBOL (If applicable)

7a. NAME OF MONITORING ORGANIZATION
Office of Naval Research

6c. ADDRESS (City, State, and ZIP Code)
Department of Computer Science
Upson Hall
Ithaca, NY 14853-7501

7b. ADDRESS (City, State, and ZIP Code)
800 N. Quincy Street
Arlington, VA 22217-5000

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
Office of Naval Research

8b. OFFICE SYMBOL (If applicable)

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
N00014-91-J-1219

8c. ADDRESS (City, State, and ZIP Code)
800 N. Quincy Street
Arlington, VA 22217-5000

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION

11. TITLE (Include Security Classification)
The Most Abstract Common Refinement

12. PERSONAL AUTHOR(S)
Limor Fix, Thomas A. Henzinger

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM ____ TO ____	94/02/04	16

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	formal specification, refinement, system composition

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We introduce a novel binary operation on specifications. The most abstract common refinement (m.a.c.r.) of two specifications P_1 and P_2 is the most abstract specification that refines both P_1 and P_2. We define the m.a.c.r.s of ω-automata and of linear temporal formulae. The m.a.c.r. operation supports a two-dimensional system design process that combines structural decomposition with stepwise refinement. As an example, we design and verify a watch in several steps, each of which simultaneously integrates and refines two partial specifications of the watch.

DTIC

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION

22a. NAME OF RESPONSIBLE INDIVIDUAL

22b. TELEPHONE (Include Area Code)

22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

The Most Abstract Common Refinement

Limor Fix* Thomas A. Henzinger†

Department of Computer Science
Cornell University
Ithaca, New York 14853

February 4, 1994

Abstract. We introduce a novel binary operation on specifications. The most abstract common refinement (m.a.c.r.) of two specifications P_1 and P_2 is the most abstract specification that refines both P_1 and P_2. We define the m.a.c.r.s of ω-automata and of linear temporal formulae. The m.a.c.r. operation supports a two-dimensional system design process that combines structural decomposition with stepwise refinement. As an example, we design and verify a watch in several steps, each of which simultaneously integrates and refines two partial specifications of the watch.

The divide-and-conquer approach to system development requires that the task of designing a large system be decomposed into subtasks. There are two avenues of decomposition that have been pursued. The *horizontal* (*static*, *structural*) *decomposition* divides the problem of designing a large system into several simpler problems of designing manageable subsystems. Horizontal decompositions of the design process are often labelled as "modular (compositional) design" [AL89, dR85, FFG91, Jon83, Lar90, Pnu85]. The *vertical* (*dynamic*, *temporal*) *decomposition* divides the problem of designing a large system into several simpler problems of gradually transforming an abstract specification into a concrete system. Vertical decompositions of the design process are often labelled as "stepwise refinement" [AL88, Bac89, CM88, Dij76, Jon89, Lyn89, WLL88, Wir71].

We present a formal framework that combines both approaches and allows the two-dimensional decomposition of the design process. Starting from a collection of system requirements, each of which describes the entire system (not just a structural component), we apply a tree-like process of simultaneously integrating and refining the initial requirements into the desired system. Each step of the process computes the so-called most abstract common refinement (m.a.c.r.) of two partial system descriptions P_1 and P_2, namely, a system description that is more concrete than both P_1 and P_2, but not unnecessarily so. The m.a.c.r. thus combines horizontal decomposition ("common") with vertical decomposition ("refinement") in a way ("most abstract") that yields a mathematically complete system design strategy.

Both parallel composition (horizontal) and standard refinement (vertical) are shown to be special cases of m.a.c.r. (*two-dimensional*): *parallel composition* combines two descriptions of system

*E-mail: fix@cs.cornell.edu. Partially supported by the Office of Naval Research under contract N00014-91-J-1219, the National Science Foundation under grant CCR-8701103, and by DARPA/NSF under grant CCR-9014363.

†E-mail: tah@cs.cornell.edu. Partially supported by the National Science Foundation under grant CCR-9200794 and by the United States Air Force Office of Scientific Research under contract F49620-93-1-0056.

parts that are given at the same level of detail; standard refinement combines two system descriptions that are given at different levels of detail; m.a.c.r., in general, combines two system descriptions, each of which describes different parts of the system at different levels of detail.

In Section 1, we define the m.a.c.r. operation on system descriptions (specifications) independent of any particular formalism. In Section 2, we show that an m.a.c.r. operation on states and actions induces an m.a.c.r. operation on behaviors (infinite sequences of states and actions) and properties (sets of behaviors). Sections 3 and 4, then, study the m.a.c.r. operation in two specific trace-based formalisms. First, we compute m.a.c.r.s of automata; then, we introduce a temporal logic with an m.a.c.r. operator. We illustrate the application of m.a.c.r.s in both formalisms by the stepwise design and verification of a watch that operates in several modes.

1 The Most Abstract Common Refinement

A *specification formalism* $(\mathcal{L}, \sqsubseteq, \bot)$ consists of

(1) the *specification language* $\mathcal{L}$—a set of specifications;
(2) the *refinement* (*satisfaction*, *implementation*) *relation* $\sqsubseteq$—a preorder on $\mathcal{L}$; and
(3) the *empty* (*inconsistent*) *specification* $\bot$—a unique bottom element of $(\mathcal{L}, \sqsubseteq)$.

Typical examples of specification formalisms are the set of process terms with simulation, the set of automata with language inclusion, and the set of linear temporal formulae with logical implication.

A specification formalism $(\mathcal{L}, \sqsubseteq, \bot)$ supports the design (refinement, implementation) of systems. Suppose that the desired properties of a system are given as a list $P_1, \ldots, P_n \in \mathcal{L}$ of requirements. The desired system, then, is a nonempty common refinement $P_{1,\ldots,n} \in \mathcal{L}$ of all n requirements:

(1) $P_{1,\ldots,n} \neq \bot$ and
(2) for all $1 \leq i \leq n$, $P_{1,\ldots,n} \sqsubseteq P_i$.

A *refinement strategy* is a procedure that, given the list of requirements, constructs a common refinement. The refinement strategy is *complete* if it finds a nonempty common refinement of the requirements whenever such a refinement exists (otherwise, the strategy returns $\bot$).

A refinement strategy is *stepwise* (*incremental*) if it constructs a common refinement of n specifications by repeatedly computing common refinements of two specifications. For example, we may first find a common refinement $P_{1,2}$ of P_1 and P_2, then a common refinement of $P_{1,2}$ and P_3, etc. A stepwise refinement strategy consists, therefore, of a control structure—a binary tree whose leaves are labelled with the initial requirements $P_1, \ldots, P_n$ and whose root is labelled with the resulting system $P_{1,\ldots,n}$—and an algorithm for computing a common refinement of two specifications. Stepwise refinement strategies have two advantages. First, the complexity of each step is more manageable than the overall problem. Second, if at a later time we h to add a new requirement P_{n+1}, the system $P_{1,\ldots,n}$ can be updated incrementally by computing a common refinement of $P_{1,\ldots,n}$ and P_{n+1}.

If a stepwise refinement strategy introduces, at any step, unnecessary constraints, then the strategy is not complete. Thus, to define complete stepwise refinement strategies, we are led to the notion of a *most abstract* (*most general*, *greatest*) *common refinement* (*m.a.c.r.*): the specification Q is an m.a.c.r. of the two specifications Q_1 and Q_2 if Q is a greatest lower bound of Q_1 and Q_2; that is,

(1) $Q \sqsubseteq Q_1$ and $Q \sqsubseteq Q_2$, and
(2) for all specifications P, if $P \sqsubseteq Q_1$ and $P \sqsubseteq Q_2$ then $P \sqsubseteq Q$.

A specification formalism $(\mathcal{L}, \sqsubseteq, \bot)$ is a *refinement structure* if every pair of specifications has an m.a.c.r. Two specifications Q_1 and Q_2 are *equivalent* if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$. Since the refinement relation $\sqsubseteq$ need not be antisymmetric, the m.a.c.r. of two specifications is unique only up to equivalence. For example, automata with language inclusion and linear temporal formulae with logical implication are refinement structures: the m.a.c.r. of two automata over a common alphabet is the product (intersection) of both automata (unique up to language equivalence); the m.a.c.r. of two formulae over a common set of variables is the conjunction of both formulae (unique up to logical equivalence).

From now on, we shall freely interpret the equality symbol between two specifications of a refinement structure as equivalence. This allows us to introduce an m.a.c.r. operation and to write $Q_1 \sqcap Q_2$ for "the" m.a.c.r. of Q_1 and Q_2. The binary function $\sqcap$ is associative, commutative, idempotent, and $Q_1 \sqsubseteq Q_2$ implies $Q_1 \sqcap Q_2 = Q_1$. It follows that refinement structures support the stepwise design of systems: every stepwise refinement strategy that, at each step, computes the m.a.c.r. of two (initial or intermediate) specifications is complete (not to mention that the computation of m.a.c.r.s eliminates the problem of "guessing" common refinements). In particular, a stepwise refinement strategy that computes m.a.c.r.s only may rely on any control structure.

We conclude by pointing out that an algorithm for computing the m.a.c.r. of two specifications provides not only a complete method for the stepwise design of systems, but also a complete method for the verification of system requirements that are given at different levels of abstraction: a system $P_1 \in \mathcal{L}$ satisfies a requirement $P_2 \in \mathcal{L}$—that is, $P_1 \sqsubseteq P_2$—iff $P_1 \sqcap P_2 = P_1$.

2 Trace refinement structures

Language inclusion is a very rough notion of refinement for trace-based formalisms. Typically one wishes to refine the states and actions of a trace by introducing new auxiliary variables [AL88], rather than throw away the entire trace. Thus we assume that the underlying alphabets of states and actions are not flat, but themselves refinement structures. We then lift these refinement structures on states and actions to a refinement structure on behaviors (infinite sequences of states and actions) and a refinement structure on properties (stutter-closed sets of behaviors). Closure under stuttering allows us to refine a single action with a finite sequence of actions [Lam83].

2.1 Uninterpreted trace refinement structures

Let $\mathcal{A} = \{a, b, c, \ldots\}$ be an alphabet of symbols. Let $\sqsubseteq$ be a binary relation on $\mathcal{A}$ and let $\bot$ be a symbol in $\mathcal{A}$ such that

(1) $(\mathcal{A}, \sqsubseteq, \bot)$ is a refinement structure and
(2) for every symbol $a \in \mathcal{A}$ there exists a *stutter symbol* $\tau_a \in \mathcal{A}$ such that for all symbols a and b, $\tau_{a \sqcap b} = \tau_a \sqcap \tau_b$.

In particular, $\tau_\bot = \bot$ for the *bottom symbol* $\bot$.

A *behavior* is an infinite sequence of symbols. We write $\alpha(i)$ for the i-th symbol of the sequence α. The behavior α_1 *refines* the behavior α_2 if all symbols of α_1 refine the corresponding symbols of α_2; that is, $\alpha_1 \sqsubseteq \alpha_2$ iff for all $i \geq 0$, $\alpha_1(i) \sqsubseteq \alpha_2(i)$.

Lemma 1 *$(\mathcal{A}^\omega, \sqsubseteq, \bot^\omega)$ is a refinement structure. In particular, for all $i \geq 0$, $(\alpha_1 \sqcap \alpha_2)(i) = \alpha_1(i) \sqcap \alpha_2(i)$.*

A *property* P is a nonempty set of behaviors that is closed under stuttering; that is, if α is in P and α' results from α by adding the stutter symbol $\tau_{\alpha(i)}$ before the i-th symbol of α, for any $i \geq 0$, then α' is also in P. We write $\mathcal{L}_{\mathcal{A}}$ for the set of properties. The property P_1 *refines* the property P_2, written $P_1 \sqsubseteq P_2$, if for every behavior α_1 in P_1 there exists a behavior α_2 in P_2 such that α_1 refines α_2.

Proposition 1 $(\mathcal{L}_{\mathcal{A}}, \sqsubseteq, \{\bot^\omega\})$ *is a refinement structure. In particular,*

$$P_1 \sqcap P_2 \;=\; \{\alpha \mid \exists \alpha_1 \in P_1 . \exists \alpha_2 \in P_2 . \alpha = \alpha_1 \sqcap \alpha_2\}.$$

We call $(\mathcal{L}_{\mathcal{A}}, \sqsubseteq, \{\bot^\omega\})$ the *trace refinement structure* of $(\mathcal{A}, \sqsubseteq, \bot)$.

2.2 The trace refinement structure of states

Let $\mathcal{U} = \{x, y, z, \ldots\}$ be a universe of variables. A *state* (V, F) consists of a set V of variables from $\mathcal{U}$ and a function F from V to a set of values, which contains the *inconsistent value* $\bot$. Given a state s, we write V_s for the variables of s and F_s for the function of s. The state s should be thought of (1) constraining, by F_s, the values of the variables in V_s, and (2) not constraining the values of the variables that are not in V_s. By Σ we denote the set of all states; by Σ_V, the set of states s with $V_s = V$.

If a variable x is assigned the inconsistent value $\bot$, this indicates that the entire state is inconsistent (cannot occur). Let $s_\bot^V = (V, \lambda x. \bot)$ be the *bottom state* on V, and $s_\bot = s_\bot^{\mathcal{U}}$. The state s *refines* the state t, written $s \sqsubseteq t$, if $V_s \supseteq V_t$ and for every variable x in V_t, if $F_s(x) \neq \bot$ then $F_s(x) = F_t(x)$. That is, s further constrains the variables of t and possibly constrains other, typically auxiliary, variables. Given a state s, the *stutter state* τ_s is s itself.

Proposition 2 $(\Sigma, \sqsubseteq, s_\bot)$ *is a refinement structure. In particular,* $V_{s \sqcap t} = V_s \cup V_t$ *and for each variable* $x \in V_{s \sqcap t}$,

$$F_{s \sqcap t}(x) \;=\; \begin{cases} \bot & \text{if } x \in V_s \text{ and } x \in V_t \text{ and } F_s(x) \neq F_t(x), \\ F_s(x) & \text{if } x \in V_s \text{ and either } x \notin V_t \text{ or } F_s(x) = F_t(x), \\ F_t(x) & \text{otherwise.} \end{cases}$$

Furthermore, for all states s *and* t, $\tau_{s \sqcap t} = \tau_s \sqcap \tau_t$.

In other words, the m.a.c.r. of two states s and t is the state that does not constrain any variables other than the variables of s and t, and those are constrained in a way that is consistent with both s and t without being unnecessarily restrictive.

If we fix the set V of variables and consider only states in Σ_V, we obtain again a refinement structure with stutter states. This is because $V_s = V_t$ implies $V_s = V_{s \sqcap t}$, and because $V_{\tau_s} = V_s$.

Corollary 1 *If* V *is a set of variables, then* $(\Sigma_V, \sqsubseteq, s_\bot^V)$ *is a refinement substructure of* $(\Sigma, \sqsubseteq, s_\bot)$. *In particular,*

$$F_{s \sqcap t}(x) \;=\; \begin{cases} \bot & \text{if } F_s(x) \neq F_t(x), \\ F_s(x) & \text{otherwise.} \end{cases}$$

It follows that all results of Section 2.1 apply for the set Σ of states, as well as for all subsets Σ_V. In particular, the state refinement structure $(\Sigma, \sqsubseteq, s_\bot)$ induces a refinement structure on state behaviors (infinite state sequences) and a trace refinement structure on state properties (stutter-closed sets of state behaviors).

2.3 The trace refinement structure of actions

An action p is a state transformation [Lam91] and can be refined in two ways, (1) by further constraining the effects of p on its variables, and (2) by introducing new auxiliary variables to model the execution of p in greater detail.

Formally, an *action* (V, R) consists of a set V of variables from $\mathcal{U}$ and a binary relation R on Σ_V such that $(s_\perp^V, s_\perp^V) \in R$. Given an action p, we write V_p for the variables of p and R_p for the relation of p. The action p should be thought of (1) constraining, by R_p, the ways in which the variables in V_p may change as a result of performing p, and (2) not constraining what happens to the variables that are not in V_p. By Π we denote the set of all actions; by Π_V, the set of actions p with $V_p = V$.

Two states s and t are *consistent* if they agree on all common variables; that is, for all $x \in V_s \cap V_t$, $F_s(x) = F_t(x)$. Let R be a binary relation on the set Σ_V of states over V, and let V' be a set of variables. The *adjustment* $R^{V'}$ of R to the variables in V' is a binary relation on $\Sigma_{V'}$ such that every pair of states in $R^{V'}$ agrees with a pair of states in R on the values of all common variables; that is, $(s', t') \in R^{V'}$ iff there exists a pair (s, t) of states in R such that s and s' are consistent, and t and t' are consistent.

Let $p_\perp^V = (V, \{(s_\perp^V, s_\perp^V)\})$ be the *bottom action* on V, and $p_\perp = p_\perp^{\mathcal{U}}$. The action p *refines* the action q, written $p \sqsubseteq q$, if $V_p \supseteq V_q$ and $R_p^{V_q} \subseteq R_q$; that is, p further constrains the variables of q and possibly constrains other, typically auxiliary, variables. Given an action p, the *stutter action* τ_p consists of the set V_p of variables and the relation $\{(s, s) | (s, t) \in R_p\}$; that is, τ_p leaves the variables of p unchanged and allows arbitrary modifications of all other variables.

Proposition 3 $(\Pi, \sqsubseteq, p_\perp)$ *is a refinement structure. In particular,* $V_{p \sqcap q} = V_p \cup V_q$ *and*

$$R_{p \sqcap q} \;=\; R_p^{V_p \cup V_q} \cap R_q^{V_p \cup V_q}.$$

Furthermore, for all actions p *and* q, $\tau_{p \sqcap q} = \tau_p \sqcap \tau_q$.

In other words, the m.a.c.r. of two actions p and q is the action that does not constrain any variables other than the variables of p and q, and those are constrained in a way that is consistent with both p and q without being unnecessarily restrictive. The action $p \sqcap q$ should, therefore, be thought of as the simultaneous concurrent execution of both p and q.

If we fix the set V of variables and consider only actions in Π_V, we obtain again a refinement structure with stutter actions. This is because $V_p = V_q$ implies $V_p = V_{p \sqcap q}$, and because $V_{\tau_p} = V_p$.

Corollary 2 *If* V *is a set of variables, then* $(\Pi_V, \sqsubseteq, p_\perp^V)$ *is a refinement substructure of* $(\Pi, \sqsubseteq, p_\perp)$. *In particular,* $R_{p \sqcap q} = R_p \cap R_q$.

It follows that all results of Section 2.1 apply for the set Π of actions, as well as for all subsets Π_V. In particular, the action refinement structure $(\Pi, \sqsubseteq, p_\perp)$ induces a refinement structure on action behaviors (infinite action sequences) and a trace refinement structure on action properties (stutter-closed sets of action behaviors).

2.4 From actions to states and back

Now we establish the connection between the state-based and the action-based view and show that they are essentially equivalent.

We use the function *Stat* to translate action properties into state properties. The *state translation* $Stat[\pi]$ of an action behavior π contains the infinite state sequence σ if for all $i \geq 0$, there

is a pair of states (s_i, t_i) in $R_{\pi(i)}$ such that $\sigma(0) = s_0$ and for $i > 0$, t_{i-1} and s_i are consistent and $\sigma(i) = t_{i-1} \sqcap s_i$.

Consider, for example, the behavior $\pi = ppqq\ldots$ with

$V_p = \{x\}$, and $(s,t) \in R_p$ iff $F_t(x) = F_s(x) + 1$;
$V_q = \{y\}$, and $(s,t) \in R_q$ iff $F_t(y) = F_s(y) + 1$;

that is, p increments the variable x and q increments the variable y. Then $Stat[\pi]$ contains a state sequence whose first four states are $(x\colon 5)$, $(x\colon 6)$, $(x\colon 7, y\colon 0)$, and $(y\colon 1)$.

The *state translation* $Stat[P]$ of an action property P is the stutter closure of the set

$$\{\sigma \mid \exists \pi \in P.\ \sigma \in Stat[\pi]\}$$

By definition, $Stat[P]$ is closed under the stuttering of states.

Proposition 4 *The state translation Stat is a homomorphism from the trace refinement structure of* $(\Pi, \sqsubseteq, p_\perp)$ *to the trace refinement structure of* $(\Sigma, \sqsubseteq, s_\perp)$.

We use the function Act to translate state properties into action properties. The *action translation* $Act[\sigma]$ of a state behavior σ is an infinite action sequence π such that for all $i \geq 0$, $V_{\pi(i)} = V_{\sigma(i)} \cup V_{\sigma(i+1)}$ and

$$\begin{aligned} R_{\pi(i)} \;=\; \{(s,t) \mid\ & \forall x \in V_{\sigma(i)}.\ F_s(x) = \perp \text{ or } F_s(x) = F_{\sigma(i)}(x) \\ & \text{and} \\ & \forall x \in V_{\sigma(i+1)}.\ F_t(x) = \perp \text{ or } F_t(x) = F_{\sigma(i+1)}(x)\}. \end{aligned}$$

The *action translation* $Act[P]$ of a state property P is the stutter closure of the set

$$\{\pi \mid \exists \sigma \in P.\ \pi = Act[\sigma]\}$$

Again by definition, $Act[P]$ is closed under the stuttering of actions.

Proposition 5 *The action translation Act is a homomorphism from the trace refinement structure of* $(\Sigma, \sqsubseteq, s_\perp)$ *to the trace refinement structure of* $(\Pi, \sqsubseteq, p_\perp)$.

3 Specification Language 1: Automata

We use Muller automata to specify properties.

A *Muller automaton* $M = (S, S^0, E, F)$ over the input alphabet $\mathcal{A}$ consists of a finite set S of control locations, a set $S^0 \subseteq S$ of start locations, a function $E : S \times S \to \mathcal{A}$ that assigns input symbols to all transitions, and a set $F \subseteq 2^S$ of acceptance sets. If $E(r, r') = a$, then the automaton can change the control location from r to r' by reading the symbol a. A behavior α is a *run* of M if there exists an infinite sequence ρ of control locations such that

(1) $\rho(0)$ is a start location ($\rho(0) \in S^0$),
(2) for all $i \geq 0$, the transition from $\rho(i)$ to $\rho(i+1)$ is labelled with $\alpha(i)$ ($E(\rho(i), \rho(i+1)) = \alpha(i)$), and
(3) the set of control locations that occur infinitely often in ρ is in F.

Let $L(M)$ be the set of runs of M.

The transition label $\perp$ indicates the absence of a transition. Given a set L of sequences, let $[L]$ be the maximal subset of L that does not contain a $\perp$ symbol. The *language* of the automaton M, then, is the set $[L(M)]$ of runs that do not contain $\perp$.

3.1 The most abstract common refinement of automata

The *property* $P(M)$ that is defined by the Muller automaton M is the stutter closure of the set $L(M)$ of runs. We write $\mathcal{L}^{\omega}_{\mathcal{A}}$ for the set of properties that are definable by Muller automata over the alphabet $\mathcal{A}$. The constructive proof of the following theorem provides a method for the stepwise design of systems from requirements that are given as Muller automata.

Theorem 1 *$\mathcal{L}^{\omega}_{\mathcal{A}}$ is closed under $\sqcap$. In particular, given two Muller automata $M_1 = (S_1, S_1^0, E_1, F_1)$ and $M_2 = (S_2, S_2^0, E_2, F_2)$,*

$$P(M_1 \sqcap M_2) = P(M_1) \sqcap P(M_2)$$

for the Muller automaton $M_1 \sqcap M_2 = (S, S^0, E, F)$ with

$$\begin{aligned} S &= S_1 \times S_2, \\ S_0 &= S_1^0 \times S_2^0, \\ E((r_1, r_2), (r_1', r_2')) &= E_1(r_1, r_1') \sqcap E_2(r_2, r_2'), \\ F &= \{R \mid R_1 \in F_1 \text{ and } R_2 \in F_2\}, \end{aligned}$$

where R_i, for $i = 1, 2$, is the i-th projection of the set R of pairs.

Corollary 3 *$(\mathcal{L}^{\omega}_{\mathcal{A}}, \sqsubseteq, \{\bot^{\omega}\})$ is a refinement substructure of the trace refinement structure of $(\mathcal{A}, \sqsubseteq, \bot)$.*

Suppose that we interpret the input symbols as actions. The m.a.c.r. $M_1 \sqcap M_2$ of two automata M_1 and M_2 performs, then, an $a \sqcap b$ action iff M_1 performs an a action and M_2 performs a b action. The $\sqcap$ operation on automata, therefore, is a generalized product operation. Indeed, standard product operations are special cases.

Most abstract common refinement as communicating composition. The communicating product M of two automata M_1 and M_2 performs an a action iff both M_1 and M_2 perform a actions; that is, $L(M) = L(M_1) \cap L(M_2)$. The communicating product is the m.a.c.r. of two automata if the underlying refinement structure $(\mathcal{A}, \sqsubseteq, \bot)$ is flat:

(1) for all actions a and b, if $a \sqsubseteq b$ then $a = b$ or $a = \bot$, and
(2) for all actions a, $\tau_a = \bot$.

Condition (1) ensures that no two distinct executable actions have an executable common refinement; condition (2) ensures that no executable action of one automaton can be performed together with a stutter action of the other automaton.

Most abstract common refinement as truly concurrent composition. The truly concurrent product of two automata M_1 and M_2 performs an (a, b) action iff M_1 performs an a action and M_2 performs a b action. The truly concurrent product is the m.a.c.r. of two automata if the underlying refinement structure $(\mathcal{A}, \sqsubseteq, \bot)$ has products:

(1) for all actions a and b, $a \sqcap b = (a, b)$,
(2) for all actions a and b, $(a, b) = \bot$ iff $a = \bot$ or $b = \bot$, and
(3) for all actions a, $\tau_a = \bot$.

Most abstract common refinement as interleaving composition. The interleaving product M of two automata M_1 and M_2 performs an a action iff either M_1 performs an a action and M_2 performs a stutter action, or vice versa. The interleaving product is the m.a.c.r. of two automata if the sets of actions $\mathcal{A}_1$ and $\mathcal{A}_2$ of the two automata are disjoint, every location contains a self-loop transition labelled with a stutter action and stutter actions label only self-loops. Finally, the underlying refinement structure has products:

(1) for all actions $a \in \mathcal{A}_1$ ($\mathcal{A}_2$) and $b \in \mathcal{A}_2$ ($\mathcal{A}_1$), if a is a stutter action then $a \sqcap b = b$,
(2) for all actions $a \in \mathcal{A}_1$ and $b \in \mathcal{A}_2$, if both a and b are not stutter actions then $a \sqcap b = \bot$.

The following example illustrates that, in general, the most abstract common refinement of two automata is perhaps best viewed not as a product, but as the "most general unifier" of the transition graphs.

3.2 Example: stepwise watch design

Consider a watch that has three modes of operation: *Display*, *Update*, and *Stopwatch*. We define the watch by three Muller automata. While each of the three automata specifies the entire watch, it provides details only for one mode of operation. The m.a.c.r. of the three automata, then, and not the product, is a suitable implementation of the watch.

The *Display* mode and its connections to the other modes are specified by the automaton $M_d = (S_d, S_d^0, E_d, F_d)$ of Figure 1 (we write $\{r\} \cup 2^S$ short for $\{\{r\} \cup R \mid R \in 2^S\}$). For simplicity we omit in Figure 1 and in the following figures self-loops of the automata labelled with stutter actions. Thus, assume that if a transition labelled a starts at location s then s has a self-loop transition labelled with τ_a. To reduce the size of M_d, we assume that the watch always displays one of only three possible time values; they are represented by the locations s_0, s_1, and s_2. Time advances with *tic* actions. The watch has two control buttons, cb_1 and cb_2. When cb_1 is pressed, the watch switches from *Display* mode to *Update* mode (location s_4), and back, via the action a_0, a_1, or a_2. The second button cb_2 causes a switch to *Stopwatch* mode (location s_3), and back, via the action b_0, b_1, or b_2. The action x abstractly represents any actions of the watch while it is not in *Display* mode.

The *Update* mode and its connections to the other modes are specified by the automaton $M_u = (S_u, S_u^0, E_u, F_u)$ of Figure 2. In locations w_0, w_1, and w_2, time advances with *tuc* actions. When cb_2 is pressed, the watch decrements time (action d). The action y abstractly represents any actions of the watch while it is not in *Update* mode (location w_3).

The *Stopwatch* mode and its connections to the other modes are specified by the automaton $M_s = (S_s, S_s^0, E_s, F_s)$ of Figure 3. In locations m_1 to m_9, time advances with *tac* actions, updating both the time and the stopwatch counter. Since the stopwatch can be initiated at any of the three possible time values, there are nine possible locations. The action z abstractly represents any actions of the watch while it is not in *Stopwatch* mode (location m_0).

In addition to the three automata we are also given a refinement relation $\sqsubseteq$ on the actions. The refinement relation for the actions of M_d and M_u is shown in Figure 4 (we omit the bottom and the stutter actions). Figure 4 also presents the most abstract common refinement $M_d \sqcap M_u$ of the two automata M_d and M_u. The m.a.c.r. automaton has a transition graph similar to the transition graph of the *Display* automaton M_d, except that location s_4, which represents the *Update* mode, has some inner structure. The transition graph of $M_d \sqcap M_u$ is also similar to the transition graph of the automaton M_u, except that location w_3 has some inner structure. The complete watch M is obtained by taking the m.a.c.r. of the two automata $M_d \sqcap M_u$ and M_s (see the full paper).

Now assume that after the completion of the watch design, we decide to produce an improved model by doubling the precision of the stopwatch component. The automaton M_s' of Figure 5 specifies the new stopwatch component. Figure 5 also shows the refinement relation for the actions of the old and the new stopwatch components. We need not construct the new watch from scratch. Rather, we first prove that M_s' refines M_s—that is, $P(M_s') \sqsubseteq P(M_s)$—and then take the m.a.c.r. of the old watch M and the new stopwatch component M_s'. To prove that $P(M_s') \sqsubseteq P(M_s)$, it

suffices to show that the two Muller automata $M_s' \sqcap M_s$ and M_s' define the same properties (trace equivalence).

4 Specification Language 2: Temporal Logic

We introduce a linear temporal logic, called $\mathrm{TL}^{\sqcap}$, to specify properties. The novelty about $\mathrm{TL}^{\sqcap}$ is that it contains an m.a.c.r. operator and therefore supports the refinement of formulae.

4.1 A temporal logic with a most-abstract-common-refinement operator

Let $2^{\mathcal{A}}$ be a set of *atomic formulae*. Every atomic formula Φ defines a set $[\![\Phi]\!]$ of symbols from $\mathcal{A}$. An atomic formula Φ is *stutter closed* if for every $a \in [\![\Phi]\!]$, τ_a is also in $[\![\Phi]\!]$.

The *temporal formulae* Ψ of $\mathrm{TL}^{\sqcap}$ are defined by the following grammar:

$$\Psi \ ::= \ \phi \mid \Box[\Phi] \mid \Box\Psi \mid \Diamond\Psi \mid \Psi_1 \sqcap \Psi_2 \mid \bot,$$

where ϕ and Φ are atomic formulae and ϕ is stutter closed. Stutter closed atomic formulae are typically used to specify the initial conditions of a system; boxed atomic formulae, to specify the transition relation; temporal operators, to specify the fairness assumptions (since $\mathrm{TL}^{\sqcap}$ lacks negation on the temporal level, both $\Box$ and $\Diamond$ are given). The m.a.c.r. operator $\sqcap$, finally, is typically used to combine several specifications of a system.

The $\mathrm{TL}^{\sqcap}$-formula Ψ defines the set $[\![\Psi]\!]$ of behaviors:

$$\begin{array}{lll}
\alpha \in [\![\phi]\!] & \text{iff} & \alpha(0) \in [\![\phi]\!]; \\
\alpha \in [\![\Box[\Phi]]\!] & \text{iff} & \text{for all } i \geq 0,\ \alpha(i) = \tau_{\alpha(i+1)} \text{ or } \alpha(i) \in [\![\Phi]\!]; \\
\alpha \in [\![\Box\Psi]\!] & \text{iff} & \text{for all } i \geq 0,\ \alpha[i..] \in [\![\Psi]\!]; \\
\alpha \in [\![\Diamond\Psi]\!] & \text{iff} & \text{for some } i \geq 0,\ \alpha[i..] \in [\![\Psi]\!]; \\
\alpha \in [\![\Psi_1 \sqcap \Psi_2]\!] & \text{iff} & \alpha \in [\![\Psi_1]\!] \sqcap [\![\Psi_2]\!]; \\
\alpha \in [\![\bot]\!] & \text{iff} & \alpha = \bot^{\omega},
\end{array}$$

where $\alpha[i..]$ denotes the infinite suffix that begins with the i-th symbol of the sequence α.

Lemma 2 *For every $TL^{\sqcap}$-formula Ψ, the set $[\![\Psi]\!]$ is a property (i.e., $[\![\Psi]\!]$ is closed under stuttering).*

A *judgment* of $\mathrm{TL}^{\sqcap}$ is an expression of the form $\Psi_1 \sqsubseteq \Psi_2$, for $\mathrm{TL}^{\sqcap}$-formulae Ψ_1 and Ψ_2. The $\mathrm{TL}^{\sqcap}$-judgment $\Psi_1 \sqsubseteq \Psi_1$ is true if $[\![\Psi_1]\!] \sqsubseteq [\![\Psi_2]\!]$. (The judgments of ordinary logic, by contrast, are of the form $\models \Phi$, which is true if the formula Φ is valid.)

$\mathrm{TL}^{\sqcap}$-judgments are used to assert correctness conditions of systems. Suppose, for example, that we have a list $\Psi_1, \ldots, \Psi_n$ of system requirements and we wish to prove that every system that satisfies these requirements also satisfies Ψ. This correctness condition is asserted by the $\mathrm{TL}^{\sqcap}$-judgment

$$\Psi_1 \sqcap \ldots \sqcap \Psi_n \sqsubseteq \Psi.$$

4.2 $\mathrm{TL}^{\sqcap}$ versus TLA

To compare $\mathrm{TL}^{\sqcap}$ with TLA [Lam91], we interpret $\mathrm{TL}^{\sqcap}$-formulae over action properties.

An *action formula* Φ is a boolean expression over a set $Var(\Phi)$ of variables that may occur in Φ either primed or unprimed. An action formula Φ is a *state formula* if only unprimed variables occur in Φ.

The action formula Φ defines the action a_Φ such that $V_{a_\Phi} = Var(\Phi)$, and $(s,t) \in R_{a_\Phi}$ iff Φ evaluates to true when every unprimed variable x is interpreted as $F_s(x)$ and every primed variable x' is interpreted as $F_t(x)$. For example, $(s,t) \in [\![y = 0 \wedge x' = x + 1]\!]$ iff $Var(s) = Var(t) = \{x, y\}$ and $F_s(y) = 0$ and $F_t(x) = F_s(x) + 1$.

The action formula Φ defines the set of actions $[\![\Phi]\!]$ that consists of the action a_Φ and if Φ is a state formula then $[\![\Phi]\!]$ also contains the stutter action τ_{a_Φ}. If the atomic formulae of $\mathrm{TL}^\sqcap$ are action formulae and the stutter closed atomic formulae of $\mathrm{TL}^\sqcap$ are state formulae, then every $\mathrm{TL}^\sqcap$-formula defines an action property. We call the resulting logic $\mathrm{TLA}^\sqcap$.

Let Ψ be a $\mathrm{TLA}^\sqcap$-formula, and let $Con[\Psi]$ be the TLA-formula that results from Ψ by replacing all m.a.c.r. operators $\sqcap$ with conjunctions $\wedge$ and by replacing all occurrences of $\bot$ with the truth value *false*. Every TLA-formula ϕ defines a state property $[\![\phi]\!]$ [Lam91]. For a $\mathrm{TLA}^\sqcap$-formula Ψ let $[\![\Psi]\!]^\sqcap$ be the adjustment of all state behaviors in $Stat[[\![\Psi]\!]]$ to the universe of variable $\mathcal{U}$. That is, if $(V_0, F_0)(V_1, F_1) \ldots \in Stat[[\![\Psi]\!]]$ then $(\mathcal{U}, F_0^{\mathcal{U}})(\mathcal{U}, F_1^{\mathcal{U}}) \ldots \in [\![\Psi]\!]^\sqcap$, where $F^{\mathcal{U}}$ is an extension of the function F to the domain $\mathcal{U}$.

Theorem 2 *For every $TLA^\sqcap$-formula Ψ, $[\![Con[\Psi]]\!] \subseteq [\![\Psi]\!]^{\mathcal{U}}$.*

The converse of this theorem is not true. Consider, for example, the universe $\mathcal{U} = \{x, y\}$ of variables and the two specifications $\Psi_1 = \Box[x' = x + 1]$ and $\Psi_2 = \Box[y' = y + 1]$. While the behavior

$$(x\colon 3, y\colon 5), (x\colon 4, y\colon 6), (x\colon 4, y\colon 7), (x\colon 4, y\colon 8), \ldots$$

is a model of the $\mathrm{TLA}^\sqcap$-formula $\Psi_1 \sqcap \Psi_2$, it is not a model of the TLA-formula $\Psi_1 \wedge \Psi_2$.

4.3 Example: verification of the watch design

In proving that property P refines property Q it is often convenient to define property P by a formula of the form $\Psi' \sqcap \Psi''$. Formula Ψ' defines only the behavior of the more refined variables of P and the formula Ψ'' set the connection among the variables Q and the variables of Ψ'. Let us consider again the watch example in Section 3.2. In Figure 6, we present a temporal formula *Stopwatch*1 that defines the first (more abstract) version of the stopwatch. The *Stopwatch*1 formula is defined over the variable t (the current time) that ranges over the values $\{0, 1, 2\}$, the variable st (the current time of the stopwatch) that ranges over the values $\{0, 1, 2\}$ and the variable m_{stw} (the mode of the watch) that ranges over the values $\{stw, nstw\}$. In Figure 6 we also present a temporal formula *Stopwatch*2 that defines the second version of the stopwatch. This formula is the m.a.c.r. of two subformulas: Ψ' refers only to the variables dt (the current time), dst (the current time of the stopwatch) and m_{stw} (the mode of the watch), where the variables dt and dst range over the values $\{0.0, 0.5, 1.0, 1.5, 2.0, 2.5\}$. The second subformula Ψ'' defines the connection between the variables t and dt and the variables st and dst.

We use the following sound rule to prove that *Stopwatch*2$\sqsubseteq$*Stopwatch*1:

$$\frac{\begin{array}{lcl} \Phi_1 \wedge \Phi_3 & \Rightarrow & \Psi_1 \\ ((\Phi_2 \vee \tau_{\Phi_2}) \wedge \Phi_3) & \Rightarrow & (\Psi_2 \vee \tau_{\Psi_2}) \\ \Phi_3 & \Rightarrow & \Psi_3 \end{array}}{(\Phi_1 \sqcap \Box[\Phi_2] \sqcap \Box\Phi_3) \sqsubseteq (\Psi_1 \sqcap \Box[\Psi_2] \sqcap \Box\Psi_3)}$$

where, Φ_3 and Ψ_3 are state formulae, $\tau_\Phi = (enable(\Phi) \wedge \bigwedge_{x \in Var(\Phi)} x = x')$[1] and $\Rightarrow$ is logical implication.

[1]The predicate *enable*(Φ) [Lam91] evaluates to *true* in state s iff there exists a state t such that Φ evaluates to *true* at (s, t).

The premises of the above rule require proving simple first order validities (see the full paper).

5 Discussion: Most Abstract Common Refinement versus Modular Refinement

Traditional stepwise refinement strategies are linear rather than tree-like: one constructs a sequence of refinement steps $P_1 \sqsupseteq P_2 \sqsupseteq \cdots \sqsupseteq P_n$ moving gradually from the most abstract specification P_1 to the most concrete, perhaps executable, specification P_n. Since the complexity of verifying a refinement step $P_{i+1} \sqsubseteq P_i$ depends on the sizes of P_i and P_{i+1}, modular refinement strategies have been proposed [Bac89, Ger89, Jon89]. The modular approach first develops (refines) system components independently, and then integrates the refined components into a single system.

The modular approach has two properties that limit its scope of applicability. Neither limitation is shared by m.a.c.r.-based refinement strategies.

First, in the modular approach, the refinement relation must be a precongruence to guarantee that the system that results from integrating the refined parts is a refinement of the initial specification. Formally, for every context $C[\cdot]$ of the specification language, $Q_1 \sqsubseteq Q_2$ must imply $C[Q_1] \sqsubseteq C[Q_2]$. Second, in the modular approach, the decomposition of a specification is driven by the structure of the specification. For example, if the top-level of a specification is a parallel-composition operator, then both processes must be developed independently.

In [WLL88] a lattice-structured refinement strategy is presented. According to this strategy, several specifications of the system are developed at different levels of abstraction and the lattice represents the refinement relation among these specifications. Our refinement strategy differs from that in [WLL88] in eliminating the problem of "guessing" common refinement, we construct a common refinement using the m.a.c.r operator.

References

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proc. of 3rd annual symposium on Logic in Computer Science*, pages 165–175. Computer Society Press, 1988.

[AL89] M. Abadi and L. Lamport. Composing specifications. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. of Rex Workshop, Stepwise Refinement of Distributed Systems- Models, Formalisms, Correctness*, pages 1–41. LNCS 430, Springer-Verlag, 1989.

[Bac89] R.J.R. Back. Refinement calculus, part ii: Parallel and reactive programs. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. of Rex Workshop, Stepwise Refinement of Distributed Systems- Models, Formalisms, Correctness*, pages 67–93. LNCS 430, Springer-Verlag, 1989.

[CM88] M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Dij76] E.W. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.

[dR85] W.P. de Roever. The quest for compositionality - a survey of proof systems for concurrency, part i. In E.J. Neuhold, editor, *IFIP working group "The role of abstract models in Computer Science*. North-Holland, 1985.

[FFG91] L. Fix, N. Francez, and O. Grumberg. Program composition and modular verification. In M. Rodriguez Artalejo J. Leach Albert, B. Monien, editor, *Proc. of 18th Colloquium on Automata, Language and Programimg*. LNCS 510, Springer-Verlag, 1991.

[Ger89] R. Gerth. Foundation of compositional program refinement- safety properties. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. of Rex Workshop, Stepwise Refinement of*

Distributed Systems- Models, Formalisms, Correctness, pages 777–808. LNCS 430, Springer-Verlag, 1989.

[Jon83] C.B. Jones. Specification and design of (parallel) programs. In R.E.A. Mason, editor, *Information Processing*, pages 321–332. Elsevier Science Publishers, North Holland, 1983.

[Jon89] B. Jonsson. On decomposing and refining specifications of distributed systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. Rex Workshop, Stepwise Refinement of Distributed Systems*, pages 361–385. LNCS 430, Springer-Verlag, 1989.

[Lam83] L. Lamport. Specifying concurrent programs modules. *ACM-TOPLAS*, 5(2):190–222, 1983.

[Lam91] L. Lamport. The temporal logic of actions. Technical Report 79, Systems Research Center, Digital Equipment Corp., Palo Alto, CA, Dec 1991.

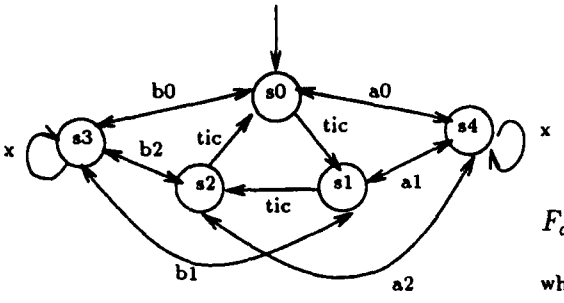
[Lar90] K.G. Larsen. Ideal specification formalism = expressivity + compositionality + decidability + testability + ..,. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR'90 Theories of Concurrency: Unification and Extension*, pages 33–56. LNCS 458, Springer-Verlag, 1990.

[Lyn89] N.A. Lynch. Multivalued possibilities mappings. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. Rex Workshop, Stepwise Refinement of Distributed Systems*, pages 519–543. LNCS 430, Springer-Verlag, 1989.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about program. In K.R. Apt, editor, *Logics and models of concurrent systems*, pages 123–144. NATO ASI Series, Vol. F13, Springer-Verlag, 1985.

[Wir71] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(1):221–227, 1971.

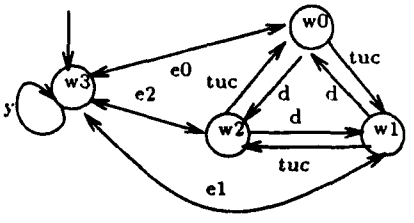
[WLL88] J.L. Welch, L. Lamport, and N. Lynch. A lattice-structured proof technique applied to a minimum spanning tree algorithm. In *Proc. of the 7th annual ACM Symposium on Principles of Distributed Computing*, pages 28–43, 1988.



$F_d = \{\{s0, s1, s2\}, \{s4\} \cup 2^W, \{s3\} \cup 2^W$

where $W = \{s0, \ldots, s4\}$

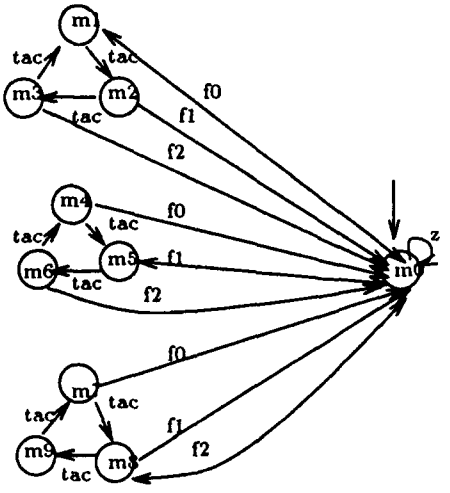
Figure 1: The M_d automaton



$F_u = \{2^W - \emptyset\}$

where $W = \{w0, w1, w2, w3\}$

Figure 2: The M_u automaton



$F_s = \{\{m1, m2, m3\}, \{m4, m5, m6\}, \{m7, m8, m9\}, \{m0\} \cup 2^W\}$

where $W = \{m0, \ldots, m9\}$

Figure 3: The M_s automaton

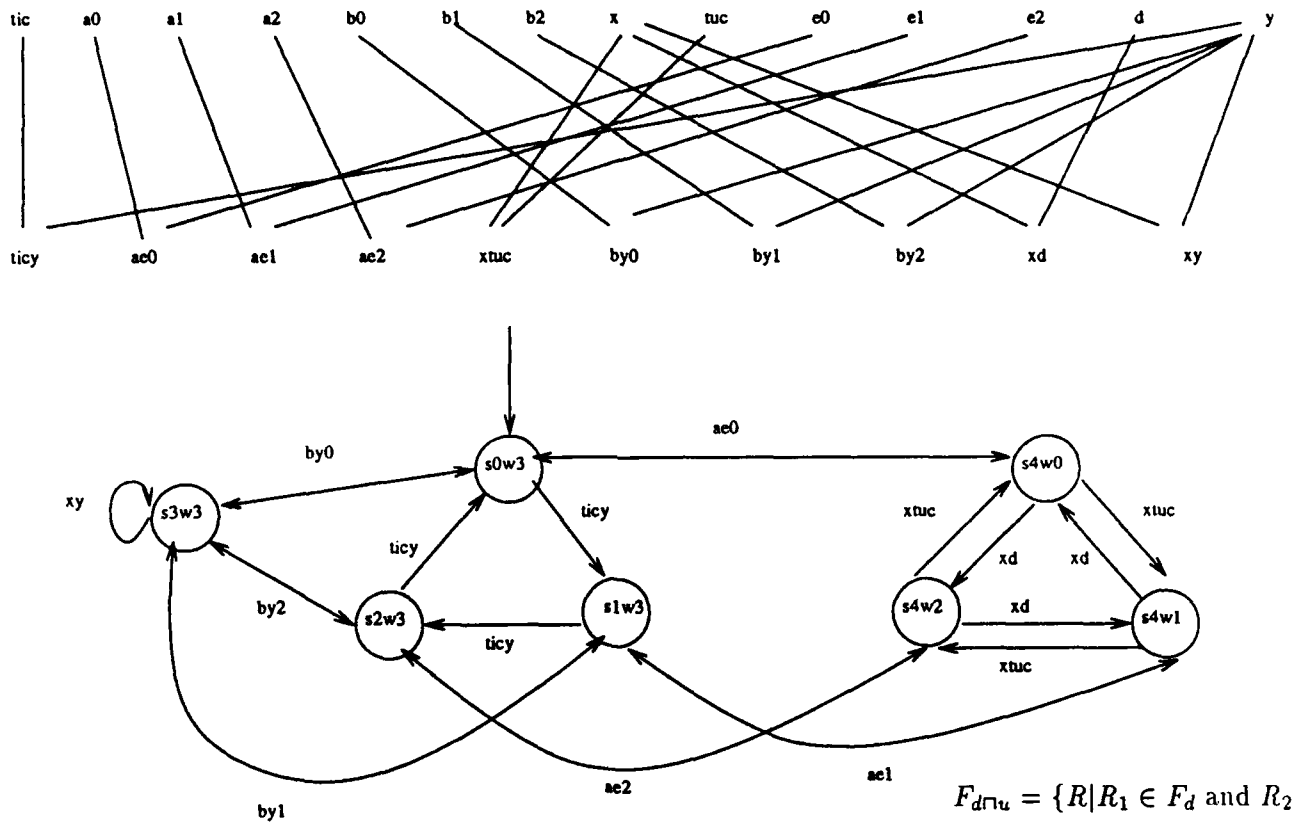


Figure 4: The refinement relation on the actions and the $M_d \sqcap M_u$ automaton

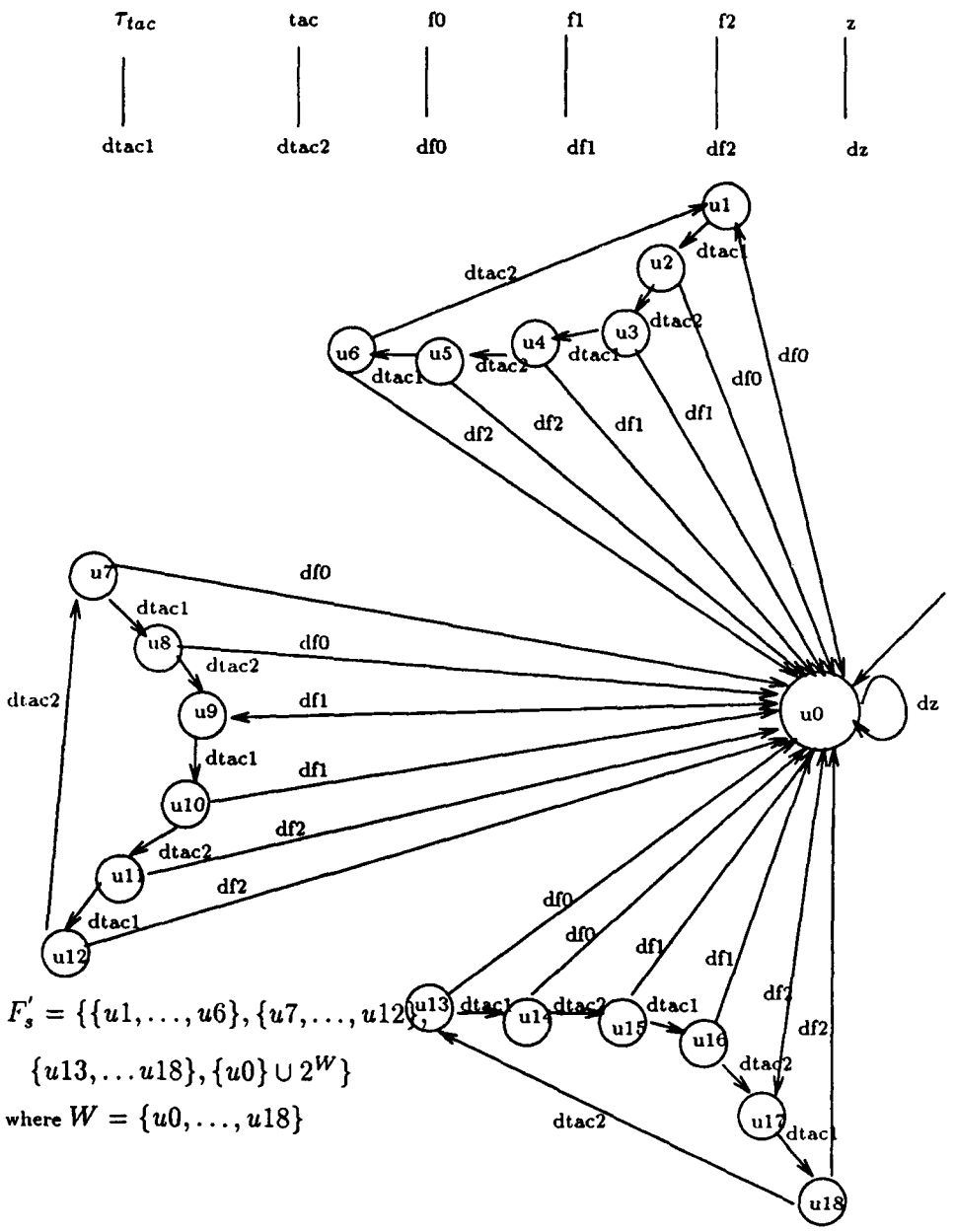


Figure 5: The refinement relation on actions and the $M_{s'}$ automaton

Stopwatch version 1:

$$
\begin{array}{lcl}
tac & = & m_{stw} = stw \wedge t' = (t+1) : mod : 3 \wedge st' = (st+1) : mod : 3 \wedge m'_{stw} = m_{stw} \\
f_0 & = & t = 0 \wedge ((m_{stw} = nstw \wedge m'_{stw} = stw \wedge st' = 0) \vee \\
 & & (m_{stw} = stw \wedge m'_{stw} = nstw \wedge st' = st)) \wedge t' = t \\
f_1 & = & t = 1 \wedge ((m_{stw} = nstw \wedge m'_{stw} = stw \wedge st' = 0)) \vee \\
 & & (m_{stw} = stw \wedge m'_{stw} = nstw \wedge st' = st)) \wedge t' = t \\
f_2 & = & t = 2 \wedge ((m_{stw} = nstw \wedge m'_{stw} = stw \wedge st' = 0)) \vee \\
 & & (m_{stw} = stw \wedge m'_{stw} = nstw \wedge st' = st)) \wedge t' = t \\
z & = & (m_{stw} = nstw) \wedge (m'_{stw} = m_{stw})
\end{array}
$$

The *Stopwatch*1 formula is:

$$
\begin{array}{l}
(t = 0 \wedge st = 0 \wedge m_{stw} = nstw) \sqcap \\
\Box[tac \vee f_0 \vee f_1 \vee f_2 \vee z]
\end{array}
$$

Stopwatch version 2:

$$
\begin{array}{lcl}
dtac1 & = & \lfloor dt \rfloor = dt \wedge dt' = (dt + 0.5) : mod : 3 \wedge dst' = (dst + 0.5) : mod : 3 \wedge m'_{stw} = m_{stw} \\
dtac2 & = & \lfloor dt \rfloor \neq dt \wedge dt' = (dt + 0.5) : mod : 3 \wedge dst' = (dst + 0.5) : mod : 3 \wedge m'_{stw} = m_{stw} \\
df_0 & = & \lfloor dt \rfloor = 0 \wedge ((m_{stw} = nstw \wedge m'_{stw} = stw \wedge dst' = 0) \vee \\
 & & (m_{stw} = stw \wedge m'_{stw} = nstw \wedge dst' = dst)) \wedge dt' = dt \\
df_1 & = & \lfloor dt \rfloor = 1 \wedge ((m_{stw} = nstw \wedge m'_{stw} = stw \wedge dst' = 0) \vee \\
 & & (m_{stw} = stw \wedge m'_{stw} = nstw \wedge dst' = dst)) \wedge dt' = dt \\
df_2 & = & \lfloor dt \rfloor = 2 \wedge ((m_{stw} = nstw \wedge m'_{stw} = stw \wedge dst' = 0) \vee \\
 & & (m_{stw} = stw \wedge m'_{stw} = nstw \wedge dst' = dst)) \wedge dt' = dt \\
dz & = & (m_{stw} = nstw) \wedge (m'_{stw} = m_{stw})
\end{array}
$$

The *Stopwatch*2 formula is: $\Psi' \sqcap \Psi''$ where

$$
\begin{array}{lcl}
\Psi' = & & (dt = 0 \wedge dst = 0 \wedge m_{stw} = nstw) \sqcap \\
 & & \Box[dtac1 \vee dtca2 \vee df_0 \vee df_1 \vee df_2 \vee dz] \\
\text{and} & & \\
\Psi'' = & & \Box(t = \lfloor dt \rfloor \wedge st = \lfloor dst \rfloor)
\end{array}
$$

Figure 6: Stopwatch versions in TLA$^{\sqcap}$